

WRITING SCIENTIFIC SOFTWARE: A GUIDE TO GOOD STYLE

The core of scientific computing is designing, writing, testing, debugging and modifying numerical software for application to a vast range of areas: from graphics, weather forecasting, and chemistry to engineering, biology, and finance. Scientists, engineers, and computer scientists need to write good, clear code, for speed, clarity, flexibility, and ease of re-use.

OLIVEIRA and STEWART provide here a guide to writing numerical software, pointing out good practices to follow, and pitfalls to avoid. By following their advice, the reader will learn how to write efficient software, and how to test it for bugs, accuracy, and performance. Techniques are explained with a variety of programming languages, and illustrated with two extensive design examples, one in Fortran 90 and one in C++, along with other examples in C, C++, Fortran 90 and Java scattered throughout the book.

Common issues in numerical computing are dealt with: for example, whether to allocate or pass “scratch” memory for temporary use, how to pass parameters to a function that is itself passed to a routine, how to allocate multidimensional arrays in C/C++/Java, and how to create suitable interfaces for routines and libraries. Advanced topics, such as recursive data structures, template programming and type binders for numerical computing, blocking and unrolling loops for efficiency, how to design software for deep memory hierarchies, and amortized doubling for efficient memory use, are also included.

This manual of scientific computing style will prove to be an essential addition to the bookshelf and lab of everyone who writes numerical software.

WRITING SCIENTIFIC SOFTWARE: A GUIDE FOR GOOD STYLE

SUELY OLIVEIRA AND DAVID E. STEWART
University of Iowa



CAMBRIDGE
UNIVERSITY PRESS

Cambridge University Press
978-0-521-67595-6 — Writing Scientific Software
Suely Oliveira , David E. Stewart
Frontmatter
[More Information](#)

CAMBRIDGE UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom
One Liberty Plaza, 20th Floor, New York, NY 10006, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
4843/24, 2nd Floor, Ansari Road, Daryaganj, Delhi - 110002, India
79 Anson Road, #06-04/06, Singapore 079906

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning and research at the highest international levels of excellence.

www.cambridge.org

Information on this title: www.cambridge.org/9780521675956

© Cambridge University Press 2006

This publication is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published 2006

A catalogue record for this publication is available from the British Library

ISBN 978-0-521-85896-0 Hardback

ISBN 978-0-521-67595-6 Paperback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

Contents

<i>Preface</i>	<i>page ix</i>
Part I Numerical Software	1
1 Why <i>numerical</i> software?	3
1.1 Efficient kernels	4
1.2 Rapid change	5
1.3 Large-scale problems	6
2 Scientific computation and numerical analysis	8
2.1 The trouble with real numbers	8
2.2 Fixed-point arithmetic	18
2.3 Algorithm stability vs. problem stability	19
2.4 Numerical accuracy and reliability	23
3 Priorities	30
3.1 Correctness	30
3.2 Numerical stability	32
3.3 Accurate discretization	32
3.4 Flexibility	33
3.5 Efficiency: time and memory	35
4 Famous disasters	36
4.1 Patriot missiles	36
4.2 Ariane 5	37
4.3 Sleipner A oil rig collapse	38
5 Exercises	39
Part II Developing Software	43
6 Basics of computer organization	45
6.1 Under the hood: what a CPU does	45
6.2 Calling routines: stacks and registers	47
6.3 Allocating variables	51
6.4 Compilers, linkers, and loaders	53

vi	<i>Contents</i>	
7	Software design	57
7.1	Software engineering	57
7.2	Software life-cycle	57
7.3	Programming in the large	59
7.4	Programming in the small	61
7.5	Programming in the middle	67
7.6	Interface design	70
7.7	Top-down and bottom-up development	75
7.8	Don't hard-wire it unnecessarily!	77
7.9	Comments	78
7.10	Documentation	80
7.11	Cross-language development	82
7.12	Modularity and all that	87
8	Data structures	90
8.1	Package your data!	90
8.2	Avoid global variables!	91
8.3	Multidimensional arrays	92
8.4	Functional representation vs. data structures	96
8.5	Functions and the “environment problem”	97
8.6	Some comments on object-oriented scientific software	106
9	Design for testing and debugging	118
9.1	Incremental testing	118
9.2	Localizing bugs	120
9.3	The mighty “print” statement	120
9.4	Get the computer to help	122
9.5	Using debuggers	129
9.6	Debugging functional representations	130
9.7	Error and exception handling	132
9.8	Compare and contrast	135
9.9	Tracking bugs	136
9.10	Stress testing and performance testing	137
9.11	Random test data	141
10	Exercises	143
	Part III Efficiency in Time, Efficiency in Memory	147
11	Be algorithm aware	149
11.1	Numerical algorithms	149
11.2	Discrete algorithms	151
11.3	Numerical algorithm design techniques	153
12	Computer architecture and efficiency	156
12.1	Caches and memory hierarchies	156

Contents

vii

12.2	A tour of the Pentium 4 TM architecture	158
12.3	Virtual memory and paging	164
12.4	Thrashing	164
12.5	Designing for memory hierarchies	165
12.6	Dynamic data structures and memory hierarchies	168
12.7	Pipelining and loop unrolling	168
12.8	Basic Linear Algebra Software (BLAS)	170
12.9	LAPACK	178
12.10	Cache-oblivious algorithms and data structures	184
12.11	Indexing vs. pointers for dynamic data structures	185
13	Global vs. local optimization	187
13.1	Picking algorithms vs. keyhole optimization	187
13.2	What optimizing compilers do	188
13.3	Helping the compiler along	191
13.4	Practicalities and asymptotic complexity	192
14	Grabbing memory when you need it	195
14.1	Dynamic memory allocation	195
14.2	Giving it back	197
14.3	Garbage collection	198
14.4	Life with garbage collection	199
14.5	Conservative garbage collection	202
14.6	Doing it yourself	203
14.7	Memory tips	205
15	Memory bugs and leaks	208
15.1	Beware: unallocated memory!	208
15.2	Beware: overwriting memory!	208
15.3	Beware: dangling pointers!	210
15.4	Beware: memory leaks!	214
15.5	Debugging tools	215
Part IV	Tools	217
16	Sources of scientific software	219
16.1	Netlib	220
16.2	BLAS	220
16.3	LAPACK	221
16.4	GAMS	221
16.5	Other sources	221
17	Unix tools	223
17.1	Automated builds: make	223
17.2	Revision control: RCS, CVS, Subversion and Bitkeeper	226

viii	<i>Contents</i>	
17.3	Profiling: prof and gprof	228
17.4	Text manipulation: grep, sed, awk, etc.	230
17.5	Other tools	232
17.6	What about Microsoft Windows?	233
Part V	Design Examples	237
18	Cubic spline function library	239
18.1	Creation and destruction	242
18.2	Output	244
18.3	Evaluation	244
18.4	Spline construction	247
18.5	Periodic splines	257
18.6	Performance testing	260
19	Multigrid algorithms	262
19.1	Discretizing partial differential equations	262
19.2	Outline of multigrid methods	264
19.3	Implementation of framework	265
19.4	Common choices for the framework	272
19.5	A first test	273
19.6	The operator interface and its uses	276
19.7	Dealing with sparse matrices	279
19.8	A second test	282
Appendix A	Review of vectors and matrices	287
A.1	Identities and inverses	288
A.2	Norms and errors	289
A.3	Errors in solving linear systems	291
Appendix B	Trademarks	292
	<i>References</i>	293
	<i>Index</i>	299

Preface

Mathematical algorithms, though usually invisible, are all around us. The micro-computer in your car controlling the fuel ignition uses a control algorithm embodying mathematical theories of dynamical systems; a Web search engine might use large-scale matrix computations; a “smart map” using a Global Positioning System to tell where you are and the best way to get home embodies numerous numerical and non-numerical algorithms; the design of modern aircraft involves simulating the aerodynamic and structural characteristics on powerful computers including supercomputers.

Behind these applications is software that does numerical computations. Often it is called scientific software, or engineering software; this software uses finite-precision floating-point (and occasionally fixed-point) numbers to represent continuous quantities.

If you are involved in writing software that does numerical computations, this book is for you. In it we try to provide tools for writing effective and efficient numerical software. If you are a numerical analyst, this book may open your eyes to software issues and techniques that are new to you. If you are a programmer, this book will explain pitfalls to avoid with floating-point arithmetic and how to get good performance without losing modern design techniques (or programming in Fortran 66). People in other areas with computing projects that involve significant numerical computation can find a bounty of useful information and techniques in this book.

But this is not a book of numerical recipes, or even a textbook for numerical analysis (numerical analysis being the study of mathematical algorithms and their behavior with finite precision floating-point arithmetic or other sources of computational errors). Nor is it a handbook on software development. It is about the development of a particular kind of software: numerical software. Several things make this kind of software a little different from other kinds of software:

- *It involves computations with floating-point numbers.* All computations with floating point arithmetic are necessarily approximate. How good are the approximations? That is the subject matter of numerical analysis. Proofs of correctness of algorithms can be irrelevant because either: (a) they completely ignore the effects of roundoff error, and so cannot identify numerical difficulties; or (b) they assume only exact properties of floating point arithmetic (floating point arithmetic is commutative $x + y = y + x$, but not associative $(x + y) + z \neq x + (y + z)$). In the latter case, they cannot prove anything useful about algorithms which are numerically accurate, but not exact (which is almost all of them).
- *It involves large-scale computations.* Large-scale computations can involve computing millions of quantities. Here efficiency is of critical importance, both in time and memory. While correctness is vital, efficiency has a special place in scientific computing. Programmers who wish to get the most out of their machine had better understand just how the hardware (and software) behind their compilers and operating systems work.
- *Requirements change rapidly.* Frequent changes in requirements and methods are a fact of life for scientific software, whether in a commercial or research environment. This means that the code had better be flexible or it will be scrapped and we will be programming from scratch again.

In every decade since the 1950s, the complexity of scientific software has increased a great deal. Object-oriented software has come to the fore in scientific and engineering software with the development of a plethora of object-oriented matrix libraries and finite element packages. Fortran used to be the clear language of choice for scientific software. That has changed. Much scientific software is now written in C, C++, Java, Matlab, Ada, and languages other than Fortran. Fortran has also changed. The Fortran 90 standard and the standards that have followed have pushed Fortran forward with many modern programming structures. But, many people who were educated on Fortran 77 or earlier versions of Fortran are unaware of these powerful new features, and of how they can be used to facilitate large-scale scientific software development. In this book when we refer to “Fortran” we will mean Fortran 90 unless another version is explicitly mentioned.

We have focused on C, C++ and Fortran 90 as the languages we know best, and are in greatest use for scientific and engineering computing. But we will also have things to say about using other languages for scientific computing, especially Java. This is not to say that other languages are not appropriate. One of the points we want to make is that many of the lessons learnt in one language can carry over to other languages, and help us to better understand the trade-offs involved in the choice of programming language and software design.

Occasionally we make historical notes about how certain systems and programming languages developed. Often important insights into the character of operating systems and programming languages, and how they are used, can be gleaned from the history of their development. It is also a useful reminder that the world of

software (including scientific software) is not static – it is changing and changing rapidly. And often our best guide to the future is a look at the past – it is certainly a good antidote to the impression often given that programming languages are eternal.

This book has been divided into five parts. The first is about what scientific and engineering software is all about, and what makes it different from conventional software design: the approximations inherent in floating-point arithmetic and other aspects of scientific software make some approaches to software development of limited applicability or irrelevant. Instead, we need to understand numerical issues much better than with other kinds of software. Also, scientific software often has much more of an emphasis on performance – there is a real need for speed. But this must be tempered by the need to keep the structure of the code from becoming “fossilized”, trying to maximize performance for some particular system. Instead we advocate a balance between flexibility and performance.

The second part is about software design. After a look at how things happen (how CPUs work, stacks and registers, variable allocation, compilers, linkers and interpreters), we emphasize practical software design and development techniques. These include incremental testing alongside some of the more practical of the “proof of correctness” ideas.

The third part is on efficiency – in both time and memory. To do this well requires a good understanding of both algorithms and computer architecture. The importance of locality is particularly emphasized. There is also a considerable amount on how to use dynamic memory allocation. This may be particularly useful for Fortran programmers who have so far avoided dynamic memory allocation.

Part IV is on tools for software development including online sources of scientific software, debuggers, and tools that have originated from the Unix operating system and have spread to many other environments.

Part V emphasizes the practicalities involved in programming scientific software. We have developed two medium-sized examples of numerical software development. One is a cubic spline library for constructing and evaluating various kinds of splines. The other is a multigrid system for the efficient iterative solution of large, sparse linear systems of equations. In these examples, the reader will see the issues discussed earlier in the context of some real examples.

As Isaac Newton said, “If I have seen far, it is because I have stood on the shoulders of giants.” We do not claim to see as far as Isaac Newton, but we have stood on the shoulders of giants. We would like to thank Barry Smith, Michael Overton, Nicholas Higham, Kendall Atkinson, and the copy-editor for their comments on our manuscript. We would especially like to thank Cambridge University Press’ technical reviewer, who was most assiduous in going through the manuscript, and whose many comments have resulted in a greatly improved manuscript. We would also like to thank the many sources of the software that we have used in the

production of this book: Microsoft (MS) Windows XP, Red-Hat Linux, the GNU compiler collection (`gcc`, `g++`, and most recently `g95`), Delorie's port of `gcc/g++` to MS Windows (`djgpp`), Minimal GNU for Windows (MinGW) and their port of `gcc/g++` to MS Windows, Intel's Fortran 90/95 compiler, the GNU tools `gmake`, `grep`, `sed`, `gdb`, etc. (many thanks to the Free Software Foundation for making these tools widely available), the `LyX` word processing software, the `MikTeX` and `TeX` implementations of `LATeX` and the DVI viewers `xdvi` and `yap`, Component Software's implementation of RCS for MS Windows, `Xfig`, `zip` and `unzip`, `WinZip`, `Valgrind`, `Octave` and `MATLAB`.